THE TREE MACHINE

AN EVALUATION OF STRATEGIES FOR REDUCING PROGRAM LOADING TIME

Pey-yun Peggy Li
and
Lennart Johnsson

Computer Science
California Institute of Technology
Pasadena CA 91125

23 May 1983

5084:TR:83

To be presented at
1983 International Conference on Parallel Processing
Bellaire, Michigan
August 23-26, 1983

The research in this report was sponsored in part
by an IBM Predoctoral Fellowship,
the Systems Development Foundation,
and the Defense Advanced Research Agency, ARPA order 3771,
monitored by the Office of Naval Research
under contract N00014-79C-0597

California Institute of Technology

THE TREE MACHINE

AN EVALUATION OF STRATEGIES FOR REDUCING PROGRAM LOADING TIME

Pey-yun Peggy Li and Lennart Johnsson

Computer Science
California Institute of Technology
Pasadena, CA. 91125

23 May 1983

ABSTRACT

The Caltech Tree Machine has an ensemble architecture. Processors are interconnected into a binary tree. Each node executes its own code. No two nodes need to execute identical code. Nodes are synchronized by messages between adjacent nodes. Since the number of nodes is intended to be large, in the order of thousands, great care needs to be exercised in devising loading strategies to make the loading time as short as possible. A constraint is also imposed by the very limited storage associated with a processor.

Nodes are assigned a type that identifies the code it shall execute. Nodes of the same type execute identical code. Tree Machine programs are frequently very regular. By exploiting this regularity, compact descriptions of the types of all nodes in the tree can be created. The limited storage of a node, and the desire to only use local information in the expansion of the compacted description implies constraints on the compression/decompression algorithms.

A loading time proportional to the height of the tree is attainable in many cases with the algorithms presented. This time is also the worst case performance for one of the algorithms. The other algorithms have a worst case performance of $O(\sqrt{N/f})$ and $O(N^{1/\log_2 f})$, where N is the total number of nodes in a tree with fanout f. The algorithms with a less favorable upper bound, in some cases allow a more compact tree description, than the algorithm with the best upper bound.

INTRODUCTION

The Caltech Tree Machine has an ensemble architecture, [Seitz 82], with processors interconnected as a binary tree. Each node is a complete, although small, von Neumann machine that can execute its own program. Synchronization is made by message passing. Hence, the Caltech Tree Machine is significantly different from other tree machine projects such as reported in [Mago 80] and [Shaw 82]. A common idea is to keep the processors small in exchange for a large number of them. The Tree Machine is an attached machine. The host compiles and loads a program into the Tree Machine, loads data into it and interacts with it during execution. The root is the only point of communication with the outside world.

The Tree Machine processor is a microprocessor designed at Caltech by Charles L. Seitz and a few students. The processor has a 16-bit data path, 16 registers, ALU, four input and four output ports, and has the control logic implemented in a PLA. It is to a large extent a Mead&Conway style design, [Mead, Conway 80], but bootstrap drivers are used, clocks are used in a non standard way ("hot clock"), and a few other techniques used to maximize the clock frequency. A major concern in the processor design was to minimize its area. The first processor, fabricated through MOSIS, the foundry service of DARPA, occupies 8 mm^2 in 4 micron technology. With a small amount of local storage two processors with associated storage will fit on a single chip in 4 micron technology, 8 in two micron technology, and at half micron technology 128 processors with storage fits on a single chip. One such chip would have an estimated capacity of 20 Mflops.

At submicron feature sizes a tree including several thousand processors is physically a small machine. It should be easy to build since the binary tree interconnection does not exhibit any wireability problem in the sense that the number of off chip interconnections can be made independent of the number of processors on a chip, [Leiserson 81]. Browning used the tree as a model for complexity analysis of several algorithms for standard problems in graph theory and matrix algebra, [Browning 80a]. A model and notation for programming the Tree Machine, and a node architecture were proposed, [Browning 79], [Browning 80b], [Browning 80c], and [Browning 81].

The programming notation and architecture for the Tree Machine has evolved from that of Browning. A version of the software system is described in [Li 81]. Brief descriptions of the current programming model and software system are given in the nexy section. Because of the very limited local storage there is a great emphasis on keeping the system software in a processor very small. A particular problem for the tree is that all communication with the outside world passes the root. For problems with a large amount of computation for each data set, this constraint is often not a severe limitation. However, when a large amount of computation is to be made maximum concurrency is sought, which implies that a large number of processors is used. The number of processors that need to be programmed can be orders of magnitude larger than the data set, [Browning 80a]. For these cases the program loading becomes a major concern. Different loading algorithms, their best and worst time complexities for regular trees, program size, and execution times in a few small sample cases is presented after the programming model is discussed.

PROGRAMMING MODEL

A node in the Tree Machine is a complete von Neumann machine. Any node can execute any piece of code that fits in its highly limited local storage. No two nodes need to have identical code to execute.

There is no global namespace. A processor can only address its local storage and its ports. Exchange of information between processors that are not nearest neighbors has to be instantiated as a sequence of communications between nearest neighbors. Processors are synchronized via message passing.

To program the tree machine, a user has to be aware of the tree structure. It is necessary to devise a tree data structure and algorithm for the problem to be solved. An algorithm has to specify what shall be computed by which processor and when those computations should occur in relation to required communication actions. The communication actions specify what information should be exchanged with which processor(s) and the type and conditions for executing the communication action. The programming notation is strongly influenced by C.A.R. Hoare's CSP, [Hoare 78], but exhibits some significant differences. For instance, processors cannot address another processor. Also, only imperative communications are allowed. This restriction simplified the port design considerably.

Data structures and algorithms can be devised for trees of arbitrary fanout. Any node can be given any fanout. The tree may have an arbitrary number of nodes. Mapping onto the binary tree configuration of the Tree Machine is performed by the software system. A fanout greater than two is achieved by introducing so called padding nodes as descendants to a Tree Machine node until the specified fanout can be realized by the descendants of the last level of padding nodes. Padding nodes are inserted in the left and right subtrees in alternating order, at every level, to minimize the unbalancing. In its pure form a padding node only serves as a communication node. Padding nodes as well as their code is generated by the software system.

A Tree Machine program has two major parts: definition of the tree structure, and definition of the programs (code and data) for all the nodes. The first part specifies the fanout of each node, and its type. The type identifies what piece of code it shall execute. All nodes of the same type execute identical code. The code is specified in the second part. The program for a node consists of a set of sequential procedures. A procedure can be evoked only by a matched message sent from the father node. A simple dispatcher, about 10 words in size, resides in each node. The dispatcher performs busy waiting for an incoming message in the input port which is connected to the father node. Whenever a new message is detected, it is dispatched to a matched procedure, which assumes the control of the processor until it terminates. The communication between nodes is asynchronous. A procedure can proceed only when and outstanding message input/output is completed successfully. A processor will always wait until an I/O instruction is completed and then proceed. A carelessly designed program may cause long waiting time for I/O completion and result in poor utilization of the nodes. The concurrency inherent in the tree may be lost to a significant extent.

Even though all nodes in the Tree Machine can execute different programs, that capability is not used in many algorithms devised so far, [Browning 80a], [Johnsson 82]. Efficient use of the machine is often obtained using only two or three node types, one for the root, one for the leaf processors, and one for the intermediate processors of the tree. Only different pieces of code, i.e., one copy of the code for each node type need to be supplied to the root. Replication of program code is easily accomplished by having the processor copy the code it receives, if it is of its type, and always pass it on to its two

descendants. The code for each node type is in the order of a few hundred words.

Compressing the definition of the tree structure and decompressing it within the tree is somewhat more complex. It is desirable to perform the decompression recursively through a series of operations based on local information only. The limited program storage limits the complexity of the decompression algorithms. The total storage for system software and floating point routines is a few hundred words. In addition to this absolute constraint there is also a trade-off between a highly explicit description that easily can be decompressed, but would take a long time to pass through the root, and a highly compressed description that would take long time to decompress. The "optimum" depends on the size and regularity of the tree.

ASSIGNING NODE TYPES IN THE TREE MACHINE

All considered algorithms load programs into tree nodes by recursively and concurrently expanding the input received by the root. A naive approach is to simply load the types of all nodes in a left to right, root to leaf order. The length of the input of node types becomes proportional to the size of the tree. This fully explicit description rapidly becomes too costly.

The algorithms that will be discussed in the following are based on the assumption that each node type has to be supplied to the root from the host at least once. The size of the tree description is at best proportional to the number of different node types, which in many cases is very small, and independent of the tree size. However, we expect that many interesting algorithms might not have the particular simple type schemes exhibited by the algorithms in [Browning 80a]. It should also be recognized that the binary tree that results from mapping an arbitrary fanout tree onto it, in many cases loses the regularity that the original tree might exhibit, Figure 1.

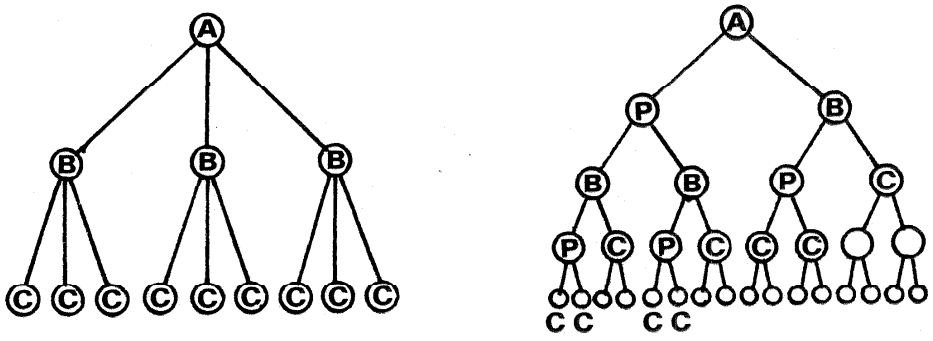


Figure 1: A three-level 3-ary tree and a corresponding binary tree

Two different assumptions on the form of the tree description supplied to the root of the Tree Machine is made:

- the host delivers a description of a binary tree, i.e., the host performs the mapping.

- the host delivers a description of the tree as defined by the programmer, i.e., the mapping onto a binary tree is performed by the Tree Machine itself.

The tree description supplied to the root is constructed by the host from the specification given in the first part of a Tree Machine program. In creating this description the host is traversing or scanning the tree in a predefined order. Two different constructs are used to compress the tree description. Nodes of identical type and fanout appearing consecutively in scan order only need to be represented once, together with a number specifying the number of occurrences. The other construct allows subtrees to be defined. These two constructs make it possible to describe very regular trees having only a few node types in an extremely compact form.

The scan order used in generating the description affects both its length and the complexity of the loading algorithm. Since the description enters the tree at the root and ripples down to the bottom, an ordering covering the tree level by level from root to leaves is prefered. At each level, nodes are assigned a binary address in left-to-right order starting from 0. Under this addressing strategy, two scan orders are studied, normal and bit-reversed, Figure 2. Normal scan order implies that nodes are visited in order of successively increasing node addresses. During bit-reversed scan order, nodes are visited in order of bit-reversed node addresses. This ordering implies that for any node its left and right subtrees are visited alternately.

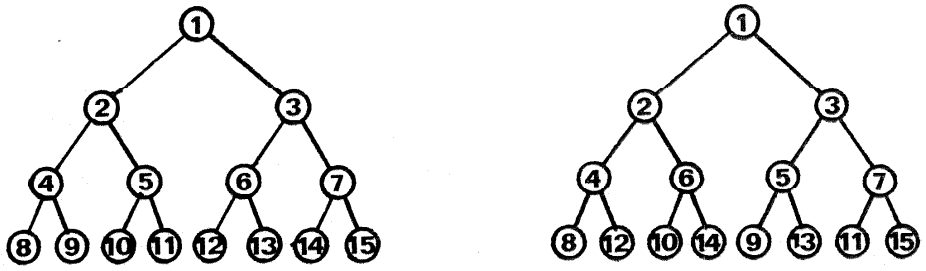


Figure 2: Normal and bit-reversed scan order

The bit-reversed ordering results in a simpler loading algorithm because a node only needs to set a flag to indicate that the next input should go to either the left or the right port. With sequential scan

order a Tree Machine node needs to monitor when the level currently being treated has been exhausted and it is time to switch to the right subtree, as well as when that is exhausted and it is time for the next level of the left subtree. The decoding of the tree description is more complex in this case.

I. Binary tree as input

For this case the host is assumed to perform the mapping onto a binary tree, if required. The padding processors are inserted in bit reversed order to keep the tree balanced. The only information provided the root by the host is a description of node types. The fanout is assumed to be two. The impact of normal and bit-reversed scan order on the loading time and the size of the down-loader program have been investigated.

Identical consecutive nodes in the given scan order are represented by a pair, (NUM, ID). NUM is the number of consecutive occurrences of nodes of type ID. The same node type may appear several times in the description. Two special delimiters, '(' and ')', are introduced to allow subtrees to be grouped together in the input. Identical subtrees encountered consecutively in scan order only need to be specified once.

The two different scan orders, in the following refered to as NO and BRO, respectively, are analyzed on three simple cases:

- Case 1. A level of the binary tree only contains nodes of one type, Figures 3 and 4.

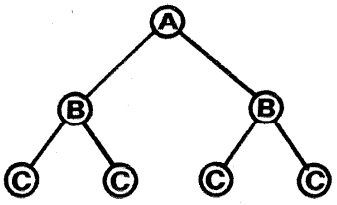
- Case 2. The binary tree is the result of mapping a two-level f-ary tree onto a binary tree. All nodes of the f-ary tre are of the same type. The necessary fanout is created by inserting padding processors in bit-reversed order to keep the tree balanced, Figures 5 and 6.

- Case 3. The binary tree is the result of mapping a k-level f-ary tree onto a binary tree. For each level padding processors are inserted as in case 2. All nodes in a level of the f-ary tree are assumed to be of the same type, Figure 7.

In the Figures 3 - 7 ',' is used as a meta symbol to separate the input entries, P is the type of padding processors, b denotes empty processors (not assigned any type), and A,B,C are the node types of the original tree.

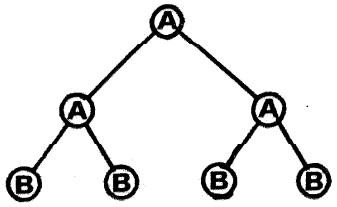
The normal and bit-reversed scan orders have the following properties:

- Case 1. A level of the binary tree only contains nodes of one type. Different levels may have different node types. The two scan orders render the same description. Its length is at best twice the number of different types, at worst twice the tree height since all nodes on a level of the tree is assumed to be of the same type, Figures 3 and 4.



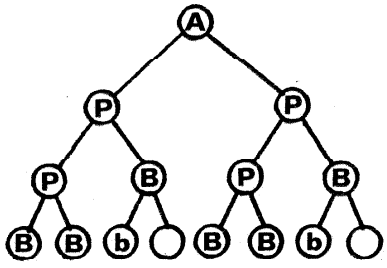
		Length
NO:	1 A, 2 B, 4 C	6
BRO:	1 A, 2 B, 4 C	6

Figure 3: A 3-level binary tree with three different node types



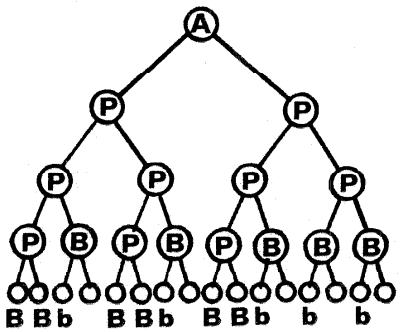
		Length
NO:	3 A, 4 B	4
BRO:	3 A, 4 B	4

Figure 4: A 3-level binary tree with two different node types



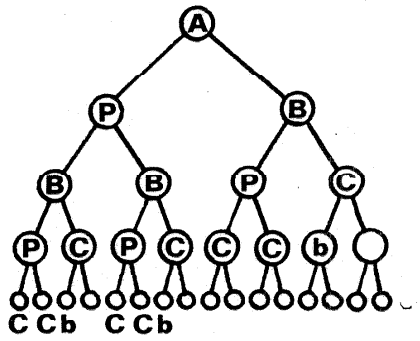
		Length
NO:	w/o subtree 1A,3P,1B,1P,3B,2b,2B	14
NO:	with subtree 1A,2(2P,3B)	9
BRO:	1A,4P,4B,2b,2B	10

Figure 5: A two-level, 6-ary tree with padding nodes



		Length
NO:	w/o subtree 1A,7P,1B,1P,1B,1P,5B,2b, 2B,2b,2B	22
NO:	with subtree 1A,2P,3(2P,3B),1(1P,2B)	18
BRO:	1A,9P,8B,5b,3B	10

Figure 6: A two-level, 11-ary tree with padding nodes



		Length
NO:	1A,1P,3(1B,1P,3C)	13
BRO:	1A,1P,2(1B,1P,2C,1b,1C), 1b,1(1B,1P,2C,1b,1C)	32

Figure 7: A three-level, 3-ary tree with padding nodes

- Case 2. The binary tree description represents the result of mapping a two level f-ary tree onto a binary tree. For simplicity of the analysis we still assume all nodes of the logical tree to be of the same type. Even with this simplifying assumption the nodes of a level of the binary tree are in general not the same, except when f equals a power of two.

  For $f \leq 10$ NO can always be written in a form such that the leaf nodes only appears once in the description. Also, under this condition the leaf nodes either appear in normal order in the binary tree, or there exist one subtree in which the nodes appear in scan order and repeated use of the subtree covers all the leaf nodes.

  For $f > 10$ the situation is not so simple. The number of occurrences of leaf nodes in the tree description increase with f. Case 1 gives a lower bound. A fairly tight upper bound is derived below.

In seeking an upper bound for the number of occurrences of leaf nodes in the binary tree description of a two-level f-ary tree with all leaf nodes being identical, it is first observed that the worst case fanout must be odd. If f is even then the f-ary tree can always be described as two identical subtrees, since padding processors are inserted in bit-reversed order. For every padding processor that is added one additional leaf node is obtained. To keep the number of leaf processors even padding processors have to be added to the left and right subtrees of the root pairwise. Use of the subtree notation then implies that such a subtree only need to be described once. If f is odd then there is one subtree instantiating an odd fanout, and one subtree representing an even fanout. They differ in fanout by one, and appear with the subtree representing the larger fanout to the left (odd or even). The subtree notation can not be used to reduce the description at this level, but there might be, and indeed are common subtrees at lower levels. The odd and even subtrees are each made up of two subtrees, recursively.

It is not difficult to see that the worst case occurs when the description of the f-ary tree can be made in terms of one subtree of odd fanout at distance two from the root, and one subtree also of odd fanout at distance three from the root, and both these subtrees have a maximum number of occurrences at their respective levels. Let $x_k = \max\{\text{leaf occurrences} \mid 2^k \leq f < 2^{k+1}\}$. Then, it follows that $x_k \leq x_{k-2} + x_{k-3}$. The initial conditions for this recurrence is $x_0 = x_1 = x_2 = 1$. There exist fanouts for which equality holds. Some such fanouts satisfy the equation $f = 8*d_{k-3}+3$ where d_{k-3} is an odd fanout for which the number of occurrences of leaf nodes in the tree description equals x_{k-3}, i.e., the maximum at that level. (Some of these d_k, but not all of them, are contained in the sequence $d_k = 2*d_{k-1} \pm 1$, $d_3 = 11$.) An upper bound for the third order recurrence equation $x_k = x_{k-2} + x_{k-3}$ is given by $\sqrt{f/2}$. The length of NO is proportional to x_k, i.e., NO is $O(\sqrt{f})$ in the worst case. However, the number of occurrences of the leaf nodes grows much slower on the average.

In BRO the number of occurrences of a leaf node equals 1 if f is a power of two, otherwise 2. These bounds are a consequence of padding nodes being inserted in bit-reversed order. Padding nodes are always encountered first. However, even though leaf nodes appear last they form one group only when f is a power of two. The reason is that leaf nodes are not allocated in bit-reversed order, but attached pairwise to the padding nodes of the previous level. With bit-riversed scan order all left descendants of the previous level are treated before any right descendant is assigned a type. Empty nodes are always encountered, except when the last level of the tree is fully populated. Therefore, BRO is of fixed length, 6 words if leaf nodes appear once in the tree description, otherwise 10 words, Figures 5 and 6.

- Case 3. The binary tree description represents the result of mapping a k-level f-ary tree onto a binary tree. All nodes at one level of the f-ary tree are assumed identical, Figure 7.

  As the number of levels in the tree grows, leaf nodes are replaced by subtrees. This replacement is easily accommodated in the description using the subtree notation. If the number of occurrences of the leaf node type in a two-level subtree is x, and the length of the descriptive sequence is s for this two-level subtree, then adding one level to the tree increases the sequence length by x*s, since each occurrence of a leaf node is replaced by s. Therefore, there are totally x^2 occurrences of the leaf node type in the resulting 3-level tree. Repeating this substitution process, the sequence length for a k-level, f-ary tree is $s(1+x+x^2+x^3+....+x^{k-2}) = s(x^{k-1}-1)/(x-1) = r$.

  For x=1, i.e., the leaf node type occurs only once in the description of a two-level f-ary tree, the sequence length is s*(k-1). It is proportional to the height of the tree. It is the minimum length possible. For x=2, the worst case for BRO, the sequence length is $s*(2^{k-1}-1)$. For x>2, s is proportional to x, which for NO is bounded by $\sqrt{f}$. The sequence length r is bounded to $\sqrt{N/f}$, where $N=f^k$, the total number of nodes in the k-level, f-ary, tree.

In conclusion, the length of both scanning orders is at best $O(\log_f N)$. The worst case for f-ary trees with all nodes at one level being identical is $O(\sqrt{N/f})$ for normal ordering, and $O(N^{1/\log_2 f})$ for bit-reversed ordering. In addition it should be noticed that there is always a minimum propagation time of $O(\log_2 N)$ from the root to the leaves

II. Arbitrary fanout tree as input

In this second case the input consists of a description of an arbitrary fanout tree. The Tree Machine nodes insert padding nodes to create the necessary fanout. Identical nodes are represented only once if they appear consecutively in the scan order, as was the case for Algorithm I. However, for this algorithm we limit the grouping of nodes to one level of the arbitrary fanout tree. The tree description will contain at least one entry per logical tree level.

An entry in the type file is of the form (NUM, ID, FANOUT), where NUM is the number of nodes of the same type and with fanout FANOUT that appears in succession in the scan order in one level of the input tree. Normal scan order is assumed. Subtrees can be defined. They are contained within '(' and ')' and can be nested.

Figure 8 shows an example of a tree description for Algorithm II.

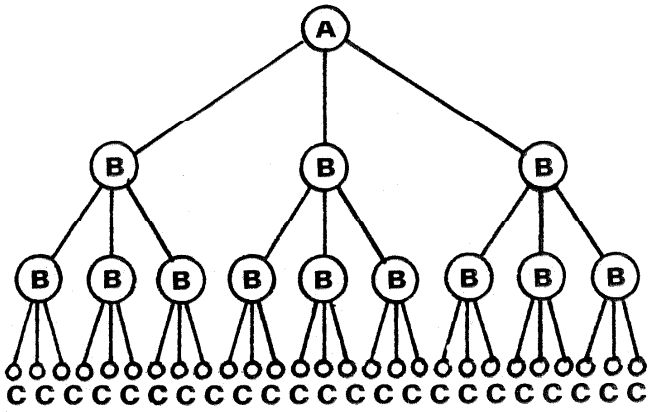


Figure 8: A 4-level, 3-ary logical tree

The basic characteristics of this algorithm are:

- A node in the Tree Machine defines, as required, one or both of its descendants to become padding nodes. Generated padding nodes may have a fanout greater than two. Every node accepts a description of an arbitrary fanout tree as input. The padding node is no different from other nodes and can generate its own padding nodes. Every node uses the first entry of the file it receives to determine its own type. One instance of the first node type is absorbed.

- A Tree Machine node has to monitor the structure of the logical tree in order to dispatch the input to proper descendants. The nodes in one level of the input tree may have different fanouts. The total number of nodes at one level of the input tree equals the sum of the fanouts of all the nodes at the preceding level. A Tree Machine node needs to keep and update information of one level of the input tree. Such information is, e.g., the total number of nodes in the input tree at the level which the current input entry is filling, the accumulated total fanouts of those nodes, the number of nodes at the current level of the input tree that shall be placed in the left and right subtrees of a Tree Machine node, and how many of those that still remains to be placed. When a logical level is filled, the processor replaces its count of the total number of nodes at the current level of the input tree by the accumulated total fanout of all nodes at that level and proceeds to treat the next level.

  In the process of dispatching one input entry, two multiplications are required to calculate the total fanouts of the nodes going to the left and to the right subtrees of a processor. The multiplication time is a significant factor in the time required by this loading algorithm.

The length of the input for this algorithm is easily seen to be $O(\log_f N)$ at best for a regular logical tree of fanout f and a total of N nodes. At worst it is O(N). The total loading time is at best $O(\log_2 N)$.

SUMMARY AND CONCLUSIONS

The loading times in instruction cycles for the three different algorithms, Algorithm I.a for binary tree input using normal scan order, Algorithm I.b for bit-reversed order, and Algorithm II for arbitrary fanout tree input, are given in Table 1. The first figure for the execution time is the time to finish processing the input at the root. The second figure is the total time elapsed until all leaves are assigned a type. The difference between these two figures corresponds to the propagation time of the last input entry. The program code is loaded immediately after downloading the tree description. The propagation delay can be reduced by pipelining the code loading phase with the node typing phase. However, the processing of an input entry in the node typing phase is much slower than processing one input word in the program loading phase. Hence, the rate of loading code is limited by the type assignment phase until that phase is completed.

Algorithm	I.a		I.b		II	
Loader size	123 words		72 words		90 words	
	input words	time inst. cyc.	input words	time inst. cyc.	input words	time inst. cyc.
Example 1	4	75/141	4	64/122	12	137/201
Example 2	18	255/471	18	220/408	27	413/621
Example 3	13	143/223	32	298/391	9	94/182
Example 4	27	269/433	164	1420/1661	15	186/370
Example 5	6	123/339	6	64/252	6	51/235
Example 6	69	627/836	10	116/306	6	51/235

Example 1. A 4-level binary tree with two different node types

Example 2. A 9-level binary tree with identical nodes in each level

Example 3. A 3-level 3-ary tree with identical nodes in each level (Figure 7)

Example 4. A 5-level 3-ary tree with identical nodes in each level

Example 5. A 2-level 28-ary tree (best case fanout for Algorithm I.a and I.b)

Example 6. A 2-level 171-ary tree (worst case fanout for Algorithm I.a, $2^7 \leq f < 2^8$)

Table 1: Comparison of Algorithms I.a, I.b, and II for some simple trees

The program sizes of Algorithms Ia:Ib:II compares as 1.7:1:1.3. The reduced program size for Algorithm II compared to Ia is a consequence of the decision to require at least one entry per level in the logical tree in Algorithm II. This constraint reduces the program complexity significantly. The program of Algorithm I.b is the shortest among the three because the bit-reversed ordering simplifies the logic of the algorithm.

Algorithm II is slower than Algorithm I.a and I.b in examples 1 and 2. Indeed Algorithm II is always the slowest of the three algorithms for binary trees. It has a longer input. Its processing at the root requires about 50% longer time than Algorithm I.b for binary trees with all nodes at a level being equal, and successive levels having different node types, Figure 9. If all nodes of the binary tree are of the same type the input length for Algorithm I.a and I.b stays constant, while that of Algorithm II is proportional to the height of the tree.

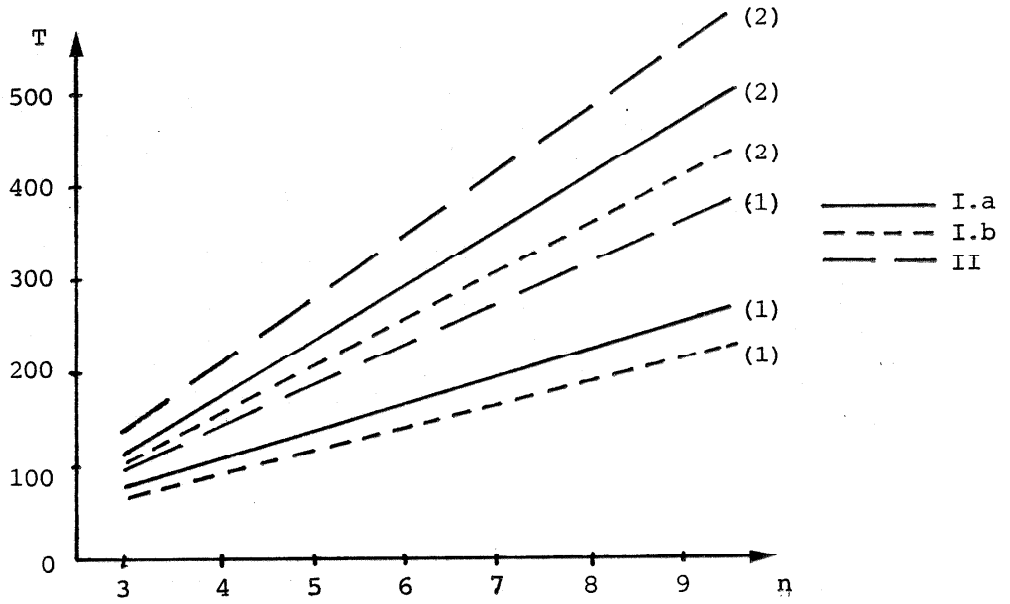


Figure 9: Instruction cycles for the root (1) and for completing the loading phase (2) for a n-level binary tree. Successive levels having different node types.

Examples 2, 4, 5, and 6, are all mapped onto a 9-level binary tree. The total number of nodes in each logical tree is 511, 121, 257 and 172 respectively. The total number of used nodes in the binary tree, including the padding nodes, is 511, 161, 511 and 341 respectively. Because of the difference of the logical tree structures, the execution time differs significantly for different algorithms. Notice in example 4(6), Algorithm I.b (Algorithm I.a) has a performance much worse than the other two algorithms. On the contrary, Algorithm II always yields a satisfactory performance for a large fanout.

The length of the input is independent of the fanout for Algorithm

II, whereas the length of the input for Algorithm I.a and I.b depends on the fanout, Figures 10 and 13. For fanouts equal to a power of two the corresponding binary tree is always balanced. The input length for Algorithm I.b as well as the total execution time is less than that of Algorithm II, except for a two level tree, Figures 9, 10 and 11. Algorithm II is always more efficient than Algorithm I.a for fanouts equal to a power of two.

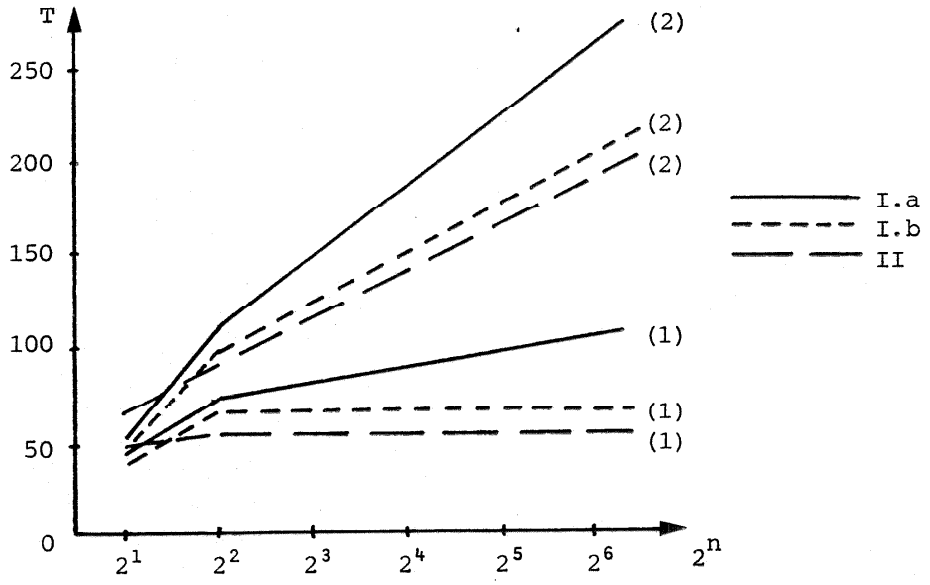


Figure 10: Instruction cycles for the root (1) and for completing the loading phase (2) for a 2-level 2^n-ary tree. Successive levels having different node types.

However, Algorithm II performs in a superior manner for most fanouts. Algorithm I.b has an exponential growth with the height of the tree, Figure 12, except for fanouts equal to a power of two, Figure 9. The exponential growth rate is independent of the fanout. The growth rate for Algorithm I.a varies from linear to exponential depending upon the fanout. It may be better or worse than Algorithm I.b, Figures 12 and 13. The growth rate of Algorithm I.a as a function of the worst case fanout in each interval, $2^n \leq f < 2^{n+1}$ follows a square root function, Figure 13.

The asymptotic loading times are summarized in Table 2. These time bounds are also bounds for the length of the input, except for Algorithms I.a and I.b, where the input can be reduced to O(1) for a binary tree with all nodes identical.

In conclusion, for a non-binary logical tree, Algorithm II has the best asymptotic properties, is also competitive for small, simple problems, and has a compact, simple code.

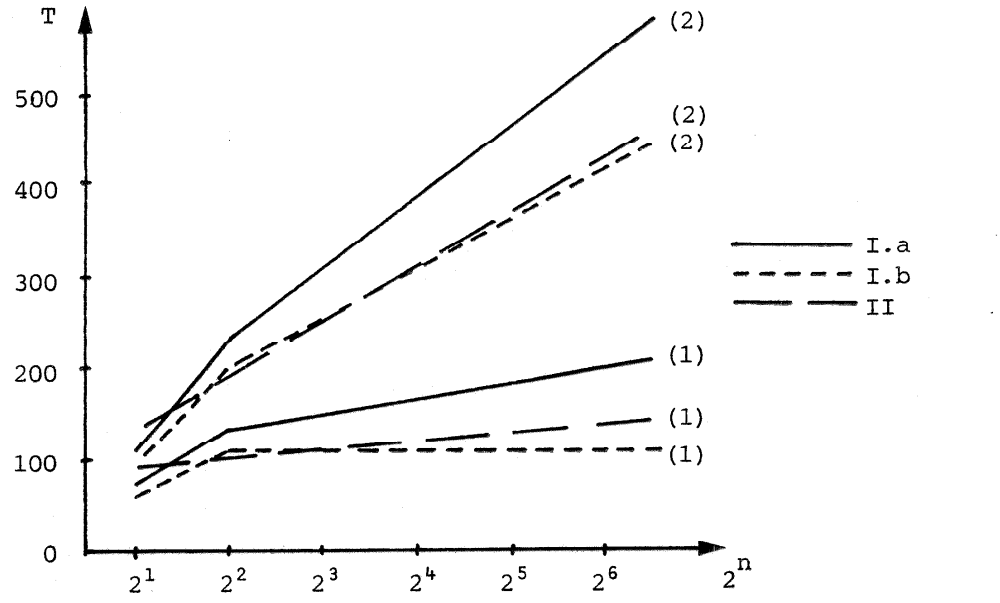


Figure 11: Instruction cycles for the root (1) and for completing the loading phase (2) for a 3-level 2^n-ary tree. Successive levels having different node types.

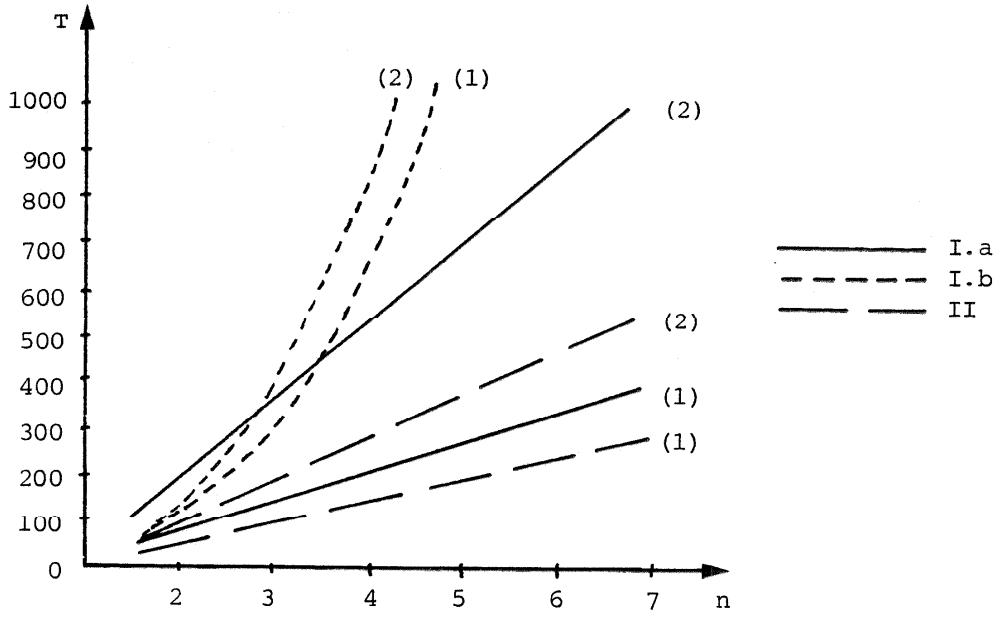


Figure 12: Instruction cycles for the root (1) and for completing the loading phase (2) for a n-level 3-ary tree. Successive levels having different node types.

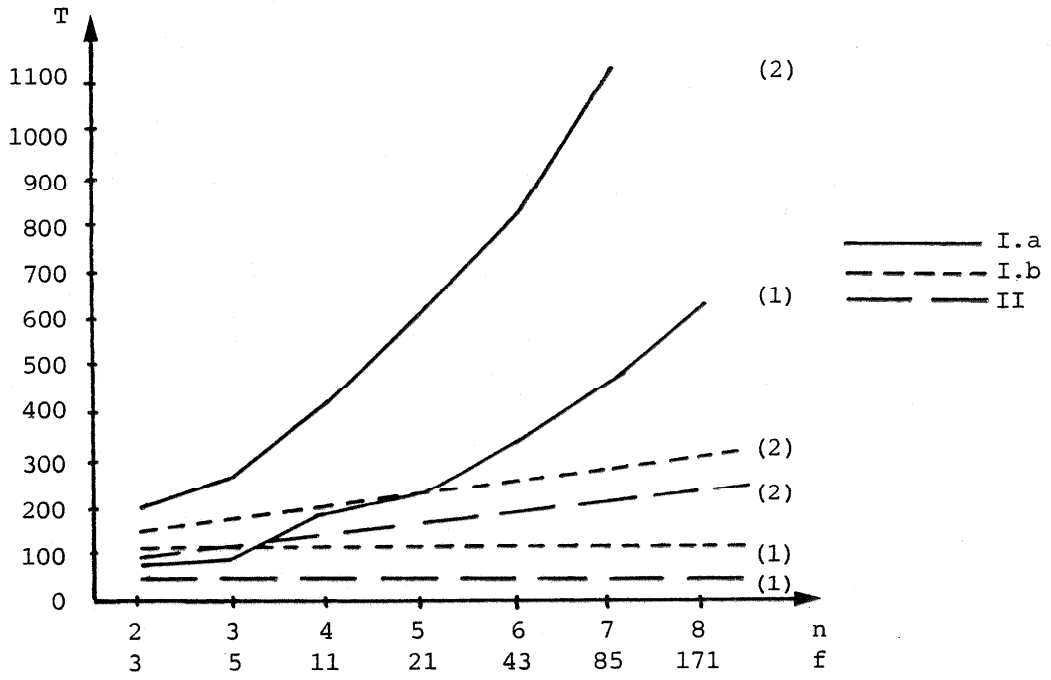


Figure 13: Instruction cycles for the root (1) and for completing the loading phase (2) for a 2-level f-ary tree.
Successive levels having different node types.

Algorithm	I.a	I.b	II
Best case	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$
Worst case	$O(\sqrt{N/f})$	$O(N^{1/\log_2 f})$	$O(\log_2 N)$

Table 2: Estimated loading times for k-level, f-ary trees, of N nodes.
All nodes of a logic level are of the same type

APPENDIX

Algorithm I.a: (Normal ordered tree description)

Data structure :

8 registers are used for temporary storage: they are

```
NUM       : stores the current input pair
ID        :
COUNT     : contains the total number of nodes at the current level
LEFT      : the number of times the current node type has been sent to the
            left subtree
RIGHT     : the number of times the current node type has been sent to the
            right subtree
REMAIN    : number of nodes that remain to be allocated at the current level
HALFCOUNT: count/2
SCOPE     : the number of nested subtrees
```

One memory word to store MY-NODE-NAME
123 momory words for the program code

Procedure :

```
(0) Initially: COUNT := 1; REMAIN := LEFT := RIGHT :=0;

(1) Read in (NUM,ID) pair;
    if ID = '(' --> go to (1);
    store ID as MY-NODE-NAME;
    decrease NUM by 1;

(2) if NUM = 0 --> read new (NUM,ID) pair;
    if NUM = 'end' or ')' --> stop;
    go to next step;

(3) if NUM <> 0 -->
      if REMAIN = 0 -->              "current level is full"
            COUNT := COUNT*2;        "start to fill the next level"
            REMAIN := COUNT;
      endif;
      if REMAIN <> 0 -->             "current level is not full"
            try to send NUM nodes with ID to  fill up the remaining empty
            nodes at the current level.  Use REMAIN, HALFCOUNT and NUM to
            decide how many should go to the left, how many to the right,
            then update LEFT, RIGHT, REMAIN and NUM.
      endif;
    endif;

(4) if NUM = 0 -->
      if LEFT <> 0 --> send (LEFT,ID) to the left subtree;
      if RIGHT <> 0 --> sent (RIGHT,ID) to the right subtree;
      if ID = '(' -->
             SCOPE := 1;
             while SCOPE <> 0 do
```

```
            read in (ID);
            send (ID) to both left and right subtrees;
            if ID = '('--> SCOPE := SCOPE+1
            else if ID = ')' --> SCOPE := SCOPE-1;
          endwhile;
      endif;
      LEFT := RIGHT := 0;
      go to (2);
```

Performance:

The time required by this algorithm is formulated in Eq.(1)

$$T = t_i+t_r+\sum_{1\le i\le k} (t_1+t_2+a_i*(t_3*(n_i-2)+t_4)+t_o+t_5*m_i)+t_p*(h-1)+t_e \qquad (1)$$

where k : total number of input pairs, one subtree is treated as one input

n_i: number of levels that the ith node name will distribute to

a_i: a boolean, which is true when $n_i>1$

m_i: the count of node name pairs in a subtree input

h : the height of the binary tree

t_i: initialization time (=4)

t_r: read in the first pair (=6 if just a node name; 15 if a subtree)

t_1: read in the next input pair (=9)

t_2: time to fill out the current level (=13 if the current level is partially filled; 8 if the current level is empty and NUM is more than the total nodes in that level; and 10 or 13 if the current level is empty and NUM is not enough to fill up the current level)

t_3: the time to fill up the next empty level (=8)

t_4: the time to fill up the last empty level for this input entry (=10 or 13)

t_o: the time to send a node pair to a descendant (=8)

t_5: the time to read in one element and send it to a subtree input (=9)

t_p: the time to propagate the last input pair one level further

t_e: the time to process the end marker

In summary, the complexity of this algorithm is in the order of $(\text{sum}(n_i)+h)$. The first term is in the order of (h) for a regular binary tree and $O(\log_2 N)$ complexity can be achieved. ($h=\log_2 N$, where N is the total number of nodes in the binary tree). The length of the input cannot be bounded to $\log_2 N$ for an irregular tree and the performance may be degraded to O(N).

Algorithm I.b: (Bit-reversed ordered tree description)

Data structure :

6 registers are used for temporary storage: they are

NUM	:	stores the current input pair
ID	:	
COUNT	:	holds the total number of nodes received so far
LEFT	:	the number of the current node type that has been sent to the left subtree
RIGHT	:	the number of the current node type that has been sent to the right subtree
SCOPE	:	the number of nested subtrees

One memory word to store MY-NODE-NAME
72 momory words for the program code

Procedure :

```
(0) Initially: COUNT := LEFT := RIGHT :=0;

(1) Read in (NUM,ID) pair;
    if ID = '(' --> go to (1);
    store ID as MY-NODE-NAME;
    decrease NUM by 1;

(2) if NUM = 0 --> read new (NUM,ID) pair;
    if NUM = 'end' or ')' --> stop;
    go to next step;

(3) if COUNT is even -->
        LEFT:=NUM/2; RIGHT:=(NUM+1)/2
    else if COUNT is odd -->
        LEFT:=(NUM+1)/2; RIGHT:=NUM/2
    endif;
    COUNT:=COUNT+NUM;

    if LEFT <> 0 --> send (LEFT,ID) to the left subtree;
    if RIGHT <> 0 --> sent (RIGHT,ID) to the right subtree;
    if ID = '(' -->
        SCOPE := 1;
        while SCOPE <> 0 do
            read in (ID);
            send (ID) to both left and right subtrees;
            if ID = '('--> SCOPE := SCOPE+1
            else if ID = ')' --> SCOPE := SCOPE-1;
        endwhile;
    endif;
    LEFT := RIGHT := 0;
    go to (2);
```

Performance:

The time required by algorithm for the bit-reversed ordering is shown in Eq.2.

$$T = t_i + \sum_{1 \le i \le k} (t_1 + t_2 + t_3 * m_i) + (h-1) * t_d + t_e \quad (2)$$

where k : the total number of input pairs. One subtree is treated as one input
h : the height of the binary tree
m_i: the count of the node type pairs in a subtree
t_i: the initialization time, including time to read in the first input pair (=8)
t_1: the time to read the next input pair (=7)
t_2: the time to process and send down one input pair (=19)
t_3: the time to read one element of a subtree and send it down (=9)
t_d: the time to propagate the last input tuple one level further (=26)
t_e: the time to process the end marker (=4)

Obviously, the performance of this algorithm is a linear function of the size of the input.

Algorithm II

Data Structure :

10 registers are used for temporary storage:

```
NUM        : stores the current tuple in these three registers
ID         :
FANOUT     :
LEFT       : the number of nodes sent to the left subtree
RIGHT      : the number of nodes sent to the right subtree
LEFTNO     : the number of nodes in the current level of the logical tree
             that remains to be sent to the left subtree
RIGHTNO    : the number of nodes in the current level of the logical tree
             that remains to be sent to the right subtree
NEWLEFTNO  : the number of nodes accumulated for the the next level of the
             logical tree that shall be sent to the left subtree
NEWRIGHTNO: the number of nodes accumulated for the next level of the
             logical tree that shall be sent to the right subtree
SCOPE      : the count of the number of nested subtrees
```

one memory word to store MY-NODE-NAME
90 memory words for program code

Procedure :

(1) Initialization: LEFT := RIGHT := 0;

(2) Read in the first tuple:

```
read in (NUM, ID, FANOUT)
"Decide how many children nodes going to the left and right"
NEWLEFTNO := (FANOUT+1) div 2;
NEWRIGHTNO := FANOUT div 2;
"skip '(' and store my ID"
if ID = '(' --> {
     NUM := FANOUT;
     read in (ID, FANOUT);
     store ID into MY-NODE-NAME;
```

(3) Generate Padding Nodes if necessary:

```
"Generate the padding nodes at the next level while the fanout is
greater than 2.   NEWLEFTNO and NEWRIGHTNO are the fanout of the new
left padding node and the new right padding node respectively. "

"Decide the padding node ID. The padding node ID has its MSB set"
ID := if MY-NODE-NAME is padId then MY-NODE-NAME else
      MY-NODE-NAME+padhead;
if NEWLEFTNO > 1 --> send (1, ID, NEWLEFTNO) to the left;
if NEWRIGHTNO > 1 --> send (1, ID, NEWRIGHTNO) to the right;
```

(4) Process next input tuple:

```
read (NUM);
"Check for the end of the input string"
if NUM = ')' or 'end' --> stop
else
     read in (ID, FANOUT);
     LEFT := RIGHT := 0;
endif;

"When the current level is full, go to the next level."
if RIGHTNO = 0 -->
       LEFTNO := NEWLEFTNO;
       RIGHTNO := NEWRIGHTNO;
       NEWLEFTNO := NEWRIGHTNO := 0;
endif;

"To fill out the remaining LEFTNO of nodes in the left half and
 the RIGHTNO of nodes in the right half with NUM of nodes of name ID."

"Update LEFT, RIGHT, LEFTNO, RIGHTNO and also accumulate the total
 fanouts in the left half and the right half of the next level and
 update NEWLEFTNO and NEWRIGHTNO. "
 NEWLEFTNO := NEWLEFTNO + LEFT * FANOUT;
 NEWRIGHTNO := NEWRIGHTNO + RIGHT * FANOUT;
```

(5) Send the node name down:

```
if LEFT <> 0 --> send (LEFT, ID, FANOUT) to the left;
if RIGHT <> 0 --> send (RIGHT, ID, FANOUT) to the right;

"If it is a subtree, read in and send out the whole subtree."
if ID = '(' -->
        SCOPE := 1;
        while SCOPE <> 0 do
               read in (ID);
               send (ID) to both left and right;
               if ID = '(' --> SCOPE := SCOPE + 1;
               if ID = ')' --> SCOPE := SCOPE - 1;
        endwhile;
endif;

"Because NUM is at most the total number of nodes at the current
 level, NUM is less than or equal to the sum of LEFTNO and
 RIGHTNO, the total unfilled nodes at the current level. Therefore,
 after process one iteration, NUM is always set to 0 and the node
 is ready to read in and process the next input tuple. "

 go to (4);
```

Performance:

For a k-level logical tree of N nodes and with p input items, $N \geq p \geq k$. Each item has fanout f_i and a repetion factor ni. Each tuple has the format:

$$(n_i, d_i, f_i),$$

where $1 \leq i \leq p$, $N = \text{sum}(n_i)$, and $1 \leq i \leq p$

The total time to load the node names is shown in Eq.3.

$$T = t_i + t_r + t_p + \underset{1 \leq i \leq p}{\text{sum}} (t_{i1} + t_{i2} * \log_2 f_i) + (h-2) * t_d + t_e \quad (3)$$

where t_i : the initialization time
t_r : the time to read in the first tuple
t_p : the time to generate padding nodes
t_{i1}: the time to process and send down the ith input tuple
t_{i2}: time to do one add-shift iteration in multiplication
h : the height of the binary tree
$h = \text{sum}(\log_2 f_{ij})$ over $i \leq j \leq k$
f_{ij} is the max. fanout at the jth logical level
t_d: the time to propagate the last input tuple one level further
t_e: the time to process the end marker

The complexity of this algorithm is of the order of max(h,p) which is caused by the third and fourth terms in Eq.3.

REFERENCES

[Browning 79]
Browning, Sally A.
Computations on a Tree of Processors.
In Proceedings of the Caltech Conference on VLSI.
Caltech Computer Science Department, January, 1979.

[Browning 80a]
Browning, S.A.
The Tree Machine: A Highly Concurrent Computing Environment.
Technical Report 3760, Computer Science, California Institute of Technology, January, 1980.

[Browning 80b]
Browning, S.A.
Algorithms for the Tree Machine.
In Introduction to VLSI Systems. Addison-Wesley, 1980.
Authors: Mead, Carver A., and Conway, Lynn A.

[Browning 80c]
Browning, S.A.
A Tree Machine.
LAMBDA 1(2):31-36, 1980.

[Browning 81]
Browning, Sally A. and Seitz, Charles L.
Communication in a Tree Machine.
In Proc. of the Second Caltech Conference on VLSI, pages 509-526. Caltech Computer Science Department, January, 1981.

[Hoare 78]
Hoare, C.A.R.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, 1978.

[Johnsson 82]
Johnsson, S. L.
Concurrent Algorithms for the Conjugate Gradient Method.
Technical Report 5040:TR:82, Computer Science, California Institute of Technology, September, 1982.

[Leiserson 81]
Leiserson, C.E.
Area-Efficient VLSI Computation.
PhD thesis, Carnegie-Mellon University, October, 1981.

[Li 81]
Li, P.
The Tree Machine Operating System.
Technical Report 4618, Computer Science, California Institute of Technology, July, 1981.

[Mago 80]
Mago,G.A.
A Cellular Computer Architecture for Functional Programming.
In Proc. Compcon Spring, 1980, pages 179-187. IEEE Computer Society, 1980.

[Mead, Conway 80]
Mead, C. A., Conway L. A.
Introduction to VLSI Systems.
Addison Wesley, 1980, .

[Seitz 82]
Seitz, C. L.
Ensemble Architectures for VLSI - A Survey and Taxonomy.
In P. Penfield Jr., editor, Proc., Conf. on Advanced Research in VLSI, pages 130 - 135. Artech House, 1982.

[Shaw 82]
Shaw, D.E.
The NON-VON Supercomputer.
Technical Report, Columbia University, August, 1982.